### Sikuli

#### Test Modularity and Reuse

##### Test Creation Language or Mechanism

Sikuli’s scripting language is Python, a high level, general purpose language which is well documented with an abundance of books, examples and tutorials. It must be noted that the version of Python used in Sikuli is not based on the standard Python interpreter, C-Python, but instead is based on the Jython interpreter. Jython is an implementation of the Python language for the Java platform. This means that user written or Open Source supplied Python/Jython modules will run if they are supported by Jython 2.5.1 (as of this writing). If any code references the C-type interface (inline C code and/or direct access to C/C++ libraries) then that code will not run in Sikuli because it is not supported by Jython. For more information see “Import user defined Python Modules” (<https://answers.launchpad.net/sikuli/+faq/1114> ) and “Language and Syntax” (<http://www.jython.org/jythonbook/en/1.0/LangSyntax.html>) for more information.

##### Function/Sub-Test Definition

Sikuli handles functions and sub-tests like any typical high level Object Oriented language. See the next section for more details.

##### Object Oriented Capability

Because the language for Sikuli automated test scripts is Python, Sikuli scripts are fully Object Oriented. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. A wealth of information may be found at <http://docs.python.org/2/tutorial/classes.html>.

#### Tool Usability

##### IDE Ease of Use

Sikuli’s IDE is rather sparse. It provides a basic text editing pane for writing and saving scripts and a chooser for selecting operator GUI actions for image capture, text input, etc. The documentation states there is a capability to use other IDE’s but this was not tested.

In order to break a project up into multiple source files, there are a few rules that must be followed to make the IDE aware of the environment. These were not difficult to find in online documentation but if the rules are not adhered to and saved in the proper places the IDE will not be able to execute the scripts.

As of this writing the IDE contains a few bugs that range from annoying to quite troublesome. The two most prevalent are:

1. Sikuli IDE Preferences - The first problem relates to directory preferences where captured images and scripts are stored. If this directory is not set **prior** to attempting to save the scripts and images, the Sikuli IDE will not properly save the data.
2. Sikuli/Java Environment - The second problem relates to the IDE not running. If a simple Xterm can be displayed from the command line, then the problem could be due to the settings for the Java environment automatically created by the Sikuli IDE. When the Sikuli IDE runs, it creates and updates a preferences file in ~userhome/.java/.userPrefs/org/sikuli/ide/pref.xml (pay attention to the “.” in .java and .userPrefs). If the Sikuli IDE is not displaying or running, then deleting that pref.xml file may be necessary before rerunning the Sikuli IDE. If that doesn’t fix the problem then deletion of the entire Sikuli directory from that path is the next step. That should fix the environment problem but doing this will remove preferences (Issue 1) and they must be set again before saving any Sikuli scripts.

Also, during this effort there was one occurrence of an anomaly where the IDE threw an internal exception and changes made since the last save were lost. Fortunately saves were done very often so there was no impact in this situation; things could have been much worse.

##### Time to Create Common Scenario

It took approximately 5 hours and 45 minutes to implement the common scenario with Sikuli - approximately 1.5 times longer than it took with Eggplant.

##### Time to Execute Common Scenario

Jenkins recorded the time to execute the scenarios with Sikuli to be:

* 2 minutes 20 seconds for successful completion of the entire scenario.
* 1 minute 20 seconds when failure was induced.

These times are about 50% slower than Eggplant execution times, and slightly faster than ATRT execution times.

#### SUT Interaction and Performance

##### Image Capture and Scan

As with the other tools, images are captured from the system under test using the mouse to click and drag a “rubber band” around the desired image. Images are saved in the user defined Sikuli directory structure as portable network graphics (png) files and the IDE allows for these files to be saved using descriptive names.

The Sikuli IDE also has a “Matching Preview” view. When activated this provides a snapshot of the currently displayed screen and allows the script writer, during script development, to determine if the captured image is found once, more than once, or not at all. It also allows for tuning of the match sensitivity, making the algorithm more or less stringent.

As with the other tools, there are capabilities to limit the search to a specific region on the screen, and to choose a “click point” or “hotspot” so that the center of the matched image is not the only place where the mouse click will take place. This is useful for instance when trying to find the correct “File” menu when there may be several visible on a screen at one time. However one of the limitations with Sikuli is in its implementation of image “hotspots”. Hotspots are limited to the view of the *captured* image. In contract, eggPlant allows the hotspot to be set *prior* to saving the image, enabling the image to be a small rectangle while still providing the capability of setting the hotspot a significant distance (in number of pixels) away from the center of the captured image.

##### Optical Character Recognition

#### Collaboration

Sikuli’s IDE generates an organized directory structure with user provided file names, and all lend themselves well to a source control system.

#### Other

##### Linking Requirements to Test Steps

This capability is not available in Sikuli.

##### Test Execution Reporting Capabilities

Sikuli’s test execution and reporting capabilities are the weakest among the evaluated tools. When running a test script, the Sikuli IDE closes its window before the script executes. After several seconds of inactivity, and what appears to be initialization time, scripted actions begin taking place. The execution time of the script is on par with the other tools, as long as the images are found. When an image is not found, there is another period of inactivity and the IDE is redisplayed, printing out where the script stopped and why. The problem is there appears to be no capability to pause or interrupt a running script, or to debug it. The script simply stops and reports either the line or function that failed.

##### Debugging

Sikuli’s default IDE is the only one evaluated that does not have a debugger. Debugging in the default IDE is possible but it relies on the creativity of the developer. In most cases it likely requires “debug” code to be placed at strategic places within the source to allow the executing script to stop as desired. At best it is cumbersome.

##### Customer Support and Licensing

Sikuli is Open Source and uses the MIT license. There is an active community of developers and users online.

##### Testing Environment and Design Considerations

Sikuli does not automatically manage VNC connections like the other tools. In eggPlant and ATRT the IDEs and test scripts provide the functionality of connecting to and executing via a VNC running on the SUT. In contrast, Sikuli does not provide a means of connecting to a remote SUT but instead simply captures what it “sees”. The obvious and easiest solution would be to install Sikuli on the SUT and run the tests from there. However our goal was to make all tools execute in the same manner, which is to run on a local host and test a remote SUT. The added benefit to this approach is it more accurately emulates the current “real world” tactical environment. After careful deliberation the following methodology was used:

For test development:

* Manually open a vncviewer on the test host (rhel1) connected to the SUT (rhel10)
* Start Sikuli on the test host (rhel1)
* Use Sikuli to manipulate the SUT (rhel10) via the vncviewer window. NOTE: Sikuli IDE displays outside of vnc - we found that this makes it easier to capture screen images for menus this way because the menus stay open in the vncviewer when we activate the Sikuli IDE for capture.

For test execution:

* Ensure that xhost + is performed on the SUT vncserver (added step to Jenkins project that starts vncserver: ssh nagios@sut “export DISPLAY=:3; xhost +” )
* Run Sikuli on test host with DISPLAY exported to SUT.
* export DISPLAY=sut:3
* /home/autotest/SikuliX/runIDE -r /home/autotest/AutomatedTest/Scenario/sikuli/Scenario.sikuli